ÉCOLE NATIONALE SUPÉRIEURE D'ÉLECTRICITÉ ET MÉCANIQUE

DEPARTEMENT GENIE INFORMATIQUE

RAPPORT DE PROJECT C

Minimal distance between cities and implementing (TSP)

Réalisé par :	Encadrant :
Mohamed Halloub Hayat Arroubi	Pr. Khalid Boukhdir

Résumé

Ce projet vise à implémenter deux algorithmes fondamentaux en informatique combinatoire : l'algorithme de Dijkstra et une méthode non heuristique pour résoudre le problème du voyageur de commerce (TSP).

Objectifs :

1. Calculer la distance minimale entre deux villes en utilisant un graphe pondéré pour représenter les villes et leurs connexions.
2. Trouver l'itinéraire optimal qui permet de visiter un ensemble de villes exactement une fois avant de revenir au point de départ, avec l'objectif de minimiser la distance totale (TSP).
3. Intégrer les deux approches pour résoudre efficacement des problèmes combinés.

Étapes du développement :

- **Représentation du graphe** : Une structure a été conçue pour modéliser les villes et leurs connexions, avec des fonctionnalités pour créer un graphe, ajouter des arêtes avec des poids spécifiques, et afficher le graphe.
- **Implémentation de Dijkstra** : L'algorithme a été utilisé pour trouver les chemins les plus courts entre deux villes données.
- **Résolution du TSP** : Une approche non heuristique (force brute ou Held-Karp) a été implémentée pour calculer l'itinéraire optimal.
- **Intégration** : L'algorithme de Dijkstra a été utilisé pour générer une matrice de distances optimales entre toutes les paires de villes, ensuite utilisée pour résoudre le TSP.

Résultats attendus :

Le programme final est capable de :

- Identifier les distances minimales entre n'importe quelles deux villes.
- Résoudre des instances du problème du voyageur de commerce pour des ensembles donnés de villes.
- Intégrer ces deux algorithmes pour répondre à des scénarios combinés de calculs d'itinéraires optimaux.

Chapitre 1 : Contexte Général du Projet

1. Introduction

Dans un monde de plus en plus connecté, les défis liés à la gestion optimale des trajets et à l'optimisation des ressources sont omniprésents. Par exemple, les entreprises de transport et de logistique doivent minimiser les distances parcourues et les coûts tout en respectant les délais imposés. Ces problématiques deviennent encore plus complexes lorsque le nombre de points à desservir augmente. Dans ce contexte, le besoin de solutions algorithmiques efficaces se fait sentir.

2.Problématique

Deux problèmes classiques se détachent :

1. Trouver le chemin le plus court entre deux points, un problème qui peut être modélisé grâce à l'algorithme de Dijkstra.
2. Résoudre le Problème du Voyageur de Commerce (TSP), qui consiste à trouver le circuit optimal permettant de visiter un ensemble de lieux tout en minimisant la distance totale parcourue.

3.Importance

En mettant en œuvre ces solutions, ce projet fournit un cadre pour la résolution de problèmes combinatoires rencontrés dans divers domaines tels que la planification des trajets, la gestion des ressources ou encore l'optimisation logistique. De plus, il constitue une illustration éducative des concepts clés en informatique théorique et appliquée.

Chapitre 2 : Conception et implémentation

➢ Structure du projet

1.Répertoire principal du projet

- Ce dossier principal contient tous les fichiers nécessaires à votre projet, notamment les fichiers sources, les en-têtes et le fichier exécutable.

2. Sous-répertoire `.vscode`

- Ce dossier contient des fichiers de configuration spécifique pour Visual Studio Code. Ces fichiers permettent de configurer des paramètres comme la compilation, l'exécution ou le débogage.

3. Fichiers sources

- **`graph.c`** :

Contient les fonctions pour manipuler les graphes, y compris la création de graphes, l'ajout d'arêtes et les fonctions nécessaires pour calculer des distances minimales avec Dijkstra.

- **`main.c`** :

Le fichier principal qui sert de point d'entrée à votre programme. Il est responsable de l'intégration des modules (graph.c, tsp.c) et de l'exécution des différentes fonctionnalités (comme afficher les résultats de Dijkstra et du TSP).

- **`tsp.c`** :

Implémente l'algorithme pour résoudre le problème du voyageur de commerce (TSP). Ce fichier contient toutes les fonctions liées au calcul de l'itinéraire optimal.

4. Fichiers d'en-tête

- **`graph.h`** :

Le fichier d'en-tête associé à `graph.c`. Il déclare les structures de données (comme les graphes, les arêtes) ainsi que les fonctions utilisées pour travailler avec les graphes.

- **`tsp.h`** :

Le fichier d'en-tête associé à `tsp.c`. Il déclare les fonctions et données nécessaires à la résolution du TSP, assurant une séparation entre le code et son interface.

5. Fichier exécutable

- **`main.exe`** :

Il s'agit du fichier binaire généré après la compilation de votre projet. Vous pouvez exécuter ce fichier pour tester les fonctionnalités de votre programme.

➢ Représentation des graphes

1.Définition et Structures de Données

Un graphe est utilisé pour modéliser les villes et les distances qui les séparent. Les éléments principaux sont :

- **Sommets** : représentant les villes.
- **Arêtes** : représentant les connexions entre deux villes avec des poids (distances).

➢ **Structures de Données utilisées**

a. Représentation d'un sommet

```c
#include<stdlib.h>
#include<stdio.h>

typedef struct Vertex {
    int id;
} Vertex;
```

`id` :

- Identifiant unique de la ville.

b. Représentation d'une arête

```
 8   typedef struct Edge {
 9       int src;
10       int dest;
11       int weight;
12   } Edge;
```

src (source) :

- Identifiant du sommet de départ (ville source).

dest (destination) :

- Identifiant du sommet d'arrivée (ville de destination).

weight :

- Poids ou coût de l'arête. Cela correspond à la distance (ou le temps de trajet, ou tout autre mesure) entre les deux villes.

c. Structure principale du graphe

```
14   typedef struct Graph {
15       Vertex* vertices;
16       Edge* edges;
17       int numVertices;
18       int numEdges;
19   } Graph;
```

vertices :

- Tableau contenant les sommets (instances de `Vertex`).
- Ce tableau permet de gérer toutes les villes présentes dans le graphe.

edges :

- Tableau contenant les arêtes (instances de `Edge`).
- Ce tableau liste toutes les connexions entre les villes avec leur poids associé.

numVertices :

- Nombre total de villes dans le graphe.
- Utilisé pour parcourir et manipuler le tableau des sommets.

numEdges :

- Nombre total de connexions (routes) dans le graphe.
- Indique combien d'entrées sont présentes dans le tableau edges.

➢ *Implémentation et Fonctions Associées au Graphe*

1. Création du Graphe

```
7   Graph* createGraph(int numVertices, int numEdges) {
8       Graph* graph = (Graph*)malloc(sizeof(Graph));
9       if (!graph) return NULL;
10
11      graph->vertices = (Vertex*)malloc(numVertices * sizeof(Vertex));
12      graph->edges = (Edge*)malloc(numEdges * sizeof(Edge));
13      graph->numVertices = numVertices;
14      graph->numEdges = numEdges;
15
16
17      for (int i = 0; i < numVertices; i++) {
18          graph->vertices[i].id = i + 1;
19      }
20
21      return graph;
22  }
```

- Cette fonction alloue dynamiquement un graphe avec un nombre défini de sommets et d'arêtes.
- Chaque sommet (ville) est identifié par un ID unique (de 1 à numVertices).
- Les sommets et les arêtes sont stockés dans des tableaux dynamiquement alloués.

2. Validation d'un Sommet

```
23  int isValidVertex(Graph* graph, int vertex) {
24      for (int i = 0; i < graph->numVertices; i++) {
25          if (graph->vertices[i].id == vertex) {
26              return 1;
27          }
28      }
29      return 0;
30  }
```

- Cette fonction vérifie si un sommet donné (identifié par son ID) existe dans le graphe.
- Elle est utilisée avant l'ajout d'arêtes pour garantir la validité des sommets source et destination.

3. Ajout d'une Arête

```
33  void addEdge(Graph* graph, int src, int dest, int weight, int index) {
34      if (index < 0 || index >= graph->numEdges) {
35          printf("Invalid edge index!\n");
36          return;
37      }
38      if (!isValidVertex(graph, src)) {
39          printf("Source vertex %d does not exist!\n", src);
40          return;
41      }
42      if (!isValidVertex(graph, dest)) {
43          printf("Destination vertex %d does not exist!\n", dest);
44          return;
45      }
46      graph->edges[index].src = src;
47      graph->edges[index].dest = dest;
48      graph->edges[index].weight = weight;
49  }
```

✓ **Description** :

- Cette fonction ajoute une arête (connexion entre deux sommets) au graphe.
- Les sommets source (`src`) et destination (`dest`) doivent être valides et existants dans le graphe.
- Les poids (`weight`) représentent les distances entre deux villes.
- Un index est utilisé pour insérer l'arête dans le tableau d'arêtes.

✓ **Gestion des erreurs** :

- Si l'index de l'arête est hors limites ou si un des sommets n'est pas valide, la fonction affiche un message d'erreur et retourne sans effectuer l'ajout.

4. Affichage du Graphe

```
50
51  void displayGraph(Graph* graph) {
52      printf("Vertices:\n");
53      for (int i = 0; i < graph->numVertices; i++) {
54          printf("City %d\n", graph->vertices[i].id);
55      }
56
57      printf("\nEdges:\n");
58      for (int i = 0; i < graph->numEdges; i++) {
59          printf("City %d -> City %d (Weight: %d)\n",
60                  graph->edges[i].src,
61                  graph->edges[i].dest,
62                  graph->edges[i].weight);
63      }
64  }
```

- Cette fonction affiche les sommets (villes) et les arêtes (connexions) du graphe.

5.Libération de Mémoire

```
66  void freeGraph(Graph* graph) {
67      if (graph) {
68          free(graph->vertices);
69          free(graph->edges);
70          free(graph);
71      }
72  }
```

- Cette fonction libère la mémoire allouée dynamiquement pour les sommets, les arêtes et la structure du graphe.

➢ Implémentation de l'Algorithme de Dijkstra

Présentation

L’algorithme de Dijkstra est utilisé pour trouver le chemin le plus court entre deux nœuds (villes) dans un graphe pondéré où les poids représentent les distances entre les nœuds. Cet algorithme est essentiel pour résoudre le problème des distances minimales entre deux villes.

Objectifs

- Calculer le chemin le plus court entre une ville source et une ville destination.
- Utiliser les poids des arêtes pour déterminer les distances minimales de manière optimale.

Code Source : Implémentation de l'algorithme de Dijkstra

```c
int dijkstra(Graph* graph, int src, int dest) {
    int* distances = (int*)malloc(graph->numVertices * sizeof(int));
    int* visited = (int*)calloc(graph->numVertices, sizeof(int));

    if (!distances || !visited) {
        printf("Memory allocation failed!\n");
        return -1;
    }


    for (int i = 0; i < graph->numVertices; i++) {
        distances[i] = INT_MAX;
    }
    distances[src - 1] = 0;

    for (int i = 0; i < graph->numVertices - 1; i++) {
        int u = -1, minDist = INT_MAX;


        for (int j = 0; j < graph->numVertices; j++) {
            if (!visited[j] && distances[j] < minDist) {
                minDist = distances[j];
                u = j;
            }
        }

        if (u == -1) break;
```

```c
        if (u == -1) break;
        visited[u] = 1;


        for (int j = 0; j < graph->numEdges; j++) {
            Edge edge = graph->edges[j];
            if (edge.src - 1 == u) {
                int v = edge.dest - 1;
                if (!visited[v] && distances[u] != INT_MAX &&
                    distances[u] + edge.weight < distances[v]) {
                    distances[v] = distances[u] + edge.weight;
                }
            }
        }
    }

    int result = distances[dest - 1];
    if (result == INT_MAX) {
        printf("No path from City %d to City %d.\n", src, dest);
        result = -1;
    }

    free(distances);
    free(visited);
    return result;
}
```

- **Initialisation** :

- Un tableau `distances` est utilisé pour stocker les distances minimales à partir du sommet source vers tous les autres sommets. Toutes les distances sont initialisées à une valeur infinie (`INT_MAX`), sauf pour la ville source, qui est initialisée à `0`.
- Un tableau `visited` permet de suivre les sommets déjà visités afin d'éviter les calculs redondants.

➢ **Recherche des sommets non visités** :
- À chaque itération, l'algorithme sélectionne le sommet non visité ayant la plus petite distance enregistrée dans `distances`.

➢ **Mise à jour des distances** :
- Pour chaque arête sortante du sommet actuel, les distances des sommets voisins sont mises à jour si une distance plus courte est trouvée via ce sommet.
- La formule utilisée est la suivante :

  Distance[v]=min(distance[v], distance[u]+poids de l'arête (u, v))

➢ **Récupération du résultat** :
- À la fin, si la distance au sommet de destination reste égale à `INT_MAX`, cela signifie qu'il n'existe aucun chemin entre la source et la destination.

➢ **Libération des ressources** :
- Les tableaux `distances` et `visited` alloués dynamiquement sont libérés pour éviter toute fuite de mémoire.

➢ Implémentation de l'Algorithme de Résolution du TSP

✓ Fonction getEdgeCost

```
51  int getEdgeCost(Graph* graph, int src, int dest) {
52      for (int i = 0; i < graph->numEdges; i++) {
53          if (graph->edges[i].src == src && graph->edges[i].dest == dest) {
54              return graph->edges[i].weight;
55          }
56      }
57      return -1;
58  }
59
```

- Cette fonction retourne le coût d'une arête (distance) entre deux sommets (`src` et `dest`) dans le graphe.

- Elle parcourt toutes les arêtes pour chercher une connexion directe.
- Retourne **-1** si aucune arête directe n'existe entre les deux sommets.

✓ Fonction Principale : TSP

```
void TSP(Graph* graph, int* path, int* fpath, long long* sum, long long* fsum, int flag, int n, int b, int a, int* s
    if (flag == n) {
        long long currentSum = 0;

        for (int i = 1; i < n; i++) {
            int src = sc[path[i - 1]];
            int dest = sc[path[i]];
            int edgeCost = getEdgeCost(graph, src, dest);

            if (edgeCost == -1) return;
            currentSum += edgeCost;
        }


        int edgeCost = getEdgeCost(graph, sc[path[n - 1]], sc[path[0]]);
        if (edgeCost == -1) return;

        currentSum += edgeCost;


        if (currentSum < *fsum) {
            *fsum = currentSum;
            for (int i = 0; i < n; i++) {
                fpath[i] = sc[path[i]];
            }
        }
```

```
            }
            return;
        }


        for (int i = a; i <= b; i++) {
            int temp = path[a];
            path[a] = path[i];
            path[i] = temp;

            TSP(graph, path, fpath, sum, fsum, flag + 1, n, b, a + 1, sc);

            temp = path[a];
            path[a] = path[i];
            path[i] = temp;
        }
    }
```

✓ **Description** :

- Cette fonction génère toutes les permutations possibles des villes et calcule leur coût pour déterminer le circuit de moindre coût.

✓ **Variables principales** :

- **`path`** : Tableau représentant une permutation des villes.
- **`fpath`** : Tableau de la meilleure permutation trouvée (itinéraire optimal).
- **`sum`** : Coût de la permutation courante.
- **`fsum`** : Meilleur coût trouvé jusqu'à présent (distance minimale).
- **`flag`** : Compteur indiquant le nombre de villes traitées dans la permutation.
- **`sc`** : Table de correspondance entre les ID de villes et leur index dans le graphe.

L'algorithme :

- Si toutes les villes sont incluses (flag == n), le coût total de l'itinéraire est calculé.
- Si cet itinéraire a un coût inférieur au précédent minimum (fsum), il est mis à jour avec le nouvel itinéraire.
- Dans le cas contraire, l'algorithme continue en générant de nouvelles permutations par échanges successifs.
- GetEdgeCost : est utilisée pour récupérer le poids (distance) des arêtes au fur et à mesure que les permutations sont générées.

Fonctionnement Étape par Étape

1. **Initialisation** :
   a. Un tableau `path` est utilisé pour générer toutes les permutations possibles des nœuds représentant les villes.
   b. Un tableau `fpath` stocke la meilleure permutation (chemin) trouvée jusqu'à présent.
2. **Calcul de Permutation** :
   a. La fonction utilise des échanges successifs des villes dans `path` pour générer toutes les permutations possibles.
3. **Évaluation des Itinéraires** :
   a. Pour chaque permutation, la fonction vérifie si toutes les arêtes sont valides (existence d'une connexion entre deux villes adjacentes).
   b. Si toutes les connexions existent, elle calcule la distance totale en ajoutant les poids de chaque connexion (y compris le retour à la ville d'origine).
4. **Mise à Jour** :
   a. Si l'itinéraire courant a un coût inférieur à celui précédemment trouvé (`fsum`), le nouveau chemin optimal (`fpath`) et son coût sont mis à jour.
5. **Fin** :

a. Après avoir exploré toutes les permutations, `fpath` contient la solution optimale et `fsum` le coût associé.

Chapitre 3 : Tester et validation

1.Test de la représentation des graphes :

```
C:\Users\hayat>cd C:\Users\hayat\Desktop\projet C

C:\Users\hayat\Desktop\projet C>gcc main.c tsp.c graph.c -o main

C:\Users\hayat\Desktop\projet C>main
Vertices:
City 1
City 2
City 3
City 4

Edges:
City 1 -> City 2 (Weight: 10)
City 1 -> City 3 (Weight: 15)
City 1 -> City 4 (Weight: 20)
City 2 -> City 1 (Weight: 10)
City 2 -> City 3 (Weight: 35)
City 2 -> City 4 (Weight: 25)
City 3 -> City 1 (Weight: 15)
City 3 -> City 2 (Weight: 35)
City 3 -> City 4 (Weight: 30)
City 4 -> City 1 (Weight: 20)
City 4 -> City 2 (Weight: 25)
City 4 -> City 3 (Weight: 30)
```

Test de displayGraph

```
C:\Users\hayat\Desktop\projet C>gcc main.c tsp.c graph.c -o main

C:\Users\hayat\Desktop\projet C>main
Vertices:
City 1
City 2
City 3
City 4

Edges:
City 1 -> City 2 (Weight: 10)
City 1 -> City 3 (Weight: 15)
City 1 -> City 4 (Weight: 20)
City 2 -> City 1 (Weight: 10)
City 2 -> City 3 (Weight: 35)
City 2 -> City 4 (Weight: 25)
City 3 -> City 1 (Weight: 15)
City 3 -> City 2 (Weight: 35)
City 3 -> City 4 (Weight: 30)
City 4 -> City 1 (Weight: 20)
City 4 -> City 2 (Weight: 25)
City 4 -> City 3 (Weight: 30)

Enter source city: 3
Enter destination city: 2
Shortest path from City 3 to City 2: 25

C:\Users\hayat\Desktop\projet C>
```

Test de l'algorithme de Dijkstra

```
C:\Users\hayat\Desktop\projet C>gcc main.c tsp.c graph.c -o main

C:\Users\hayat\Desktop\projet C>main
Vertices:
City 1
City 2
City 3
City 4

Edges:
City 1 -> City 2 (Weight: 10)
City 1 -> City 3 (Weight: 15)
City 1 -> City 4 (Weight: 20)
City 2 -> City 1 (Weight: 10)
City 2 -> City 3 (Weight: 35)
City 2 -> City 4 (Weight: 25)
City 3 -> City 1 (Weight: 15)
City 3 -> City 2 (Weight: 35)
City 3 -> City 4 (Weight: 30)
City 4 -> City 1 (Weight: 20)
City 4 -> City 2 (Weight: 25)
City 4 -> City 3 (Weight: 30)

Optimal Route:
City 1 -> City 2 -> City 4 -> City 3 -> City 1
Minimum Cost: 80

C:\Users\hayat\Desktop\projet C>
```

Test de l'algorithme de résolution du TSP

2.Évaluation des résultats:

a-Fonctionnement Global:

Le programme fonctionne de manière satisfaisante pour la création et l'affichage d'un graphe. Les sommets et les arêtes sont bien représentés avec leurs poids correspondants. Cela témoigne d'une bonne implémentation des structures de données, notamment les tableaux dynamiques pour les sommets (vertices) et les arêtes (edges).

b-Sommets (Vertices) :

Les sommets du graphe sont bien affichés. Chaque ville est identifiée de manière unique, ce qui permet une représentation claire et organisée des nœuds. Cela valide l'initialisation correcte de la structure de graphe, où chaque sommet est assigné un identifiant unique.

c-Arêtes (Edges) :

Les connexions entre les villes sont bien affichées avec leurs distances (poids), et l'information affichée est cohérente avec les données définies. Cependant, une duplication des arêtes est visible dans la sortie : Chaque connexion est affichée plusieurs fois, ce qui peut perturber l'interprétation des résultats. Cela indique une optimisation possible au niveau de la fonction addEdge ou displayGraph.

d-Validation de la Création du Graphe :

Structure correcte : La représentation montre que le graphe est correctement structuré avec une liste d'arêtes, chaque arête étant associée à une paire de sommets et à un poids.
Connexions complètes : Toutes les connexions semblent bien prises en compte (chaque paire source-destination est affichée avec le poids correct). Cela témoigne du bon fonctionnement de l'ajout des arêtes.

f-Fonctionnement de l'Algorithme de Dijkstra :

Le résultat montre que l'algorithme de Dijkstra implémenté est efficace pour : Identifier les sommets adjacents à partir d'un sommet donné. Mettre à jour les distances minimales correctement grâce à la structure de graphe mise en place. Répondre avec le chemin le plus court lorsqu'il existe.

g-Évaluation du code TSP :

- ✓ **Représentation du graphe :**
  - Le code définit 4 villes (Ville 1, Ville 2, Ville 3, Ville 4).
  - Il répertorie les arêtes (connexions) entre les villes et les poids (distances) correspondants.
- ✓ **Itinéraire optimal :**
  - Le code affiche l'itinéraire optimal trouvé par l'algorithme TSP : Ville 1 -> Ville 2 -> Ville 4 -> Ville 3 -> Ville 1.

- ✓ Coût minimum :
    - Le coût total minimum de l'itinéraire optimal est de 80.

Dans l'ensemble, le code démontre la mise en œuvre réussie d'un algorithme TSP qui trouve l'itinéraire optimal et le coût minimum pour le graphe de villes donné. La représentation du graphe et les résultats affichés semblent corrects.

Chapitre 4 : Intégration de Dijkstra et du TSP

1.Fonction createDistanceMatrix :

```
int** createDistanceMatrix(Graph* graph) {
    int n = graph->numVertices;
    int** distanceMatrix = (int**)malloc(n * sizeof(int*));
    for (int i = 0; i < n; i++) {
        distanceMatrix[i] = (int*)malloc(n * sizeof(int));
    }

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                distanceMatrix[i][j] = 0;
            } else {
                distanceMatrix[i][j] = dijkstra(graph, i + 1, j + 1);
            }
        }
    }
```

✓ **Description :**

Le but de cette fonction est de créer une matrice de taille n×n , où n représente la taille du graphe, contenant les distances optimales entre la ville i et la ville j. Ce programme utilise l'algorithme de Dijkstra pour déterminer la distance entre deux villes, et deux boucles sont utilisées pour remplir la matrice.

2.Fonction freeDistanceMatrix :

```
91  void freeDistanceMatrix(int** distanceMatrix, int n) {
92      for (int i = 0; i < n; i++) {
93          free(distanceMatrix[i]);
94      }
95      free(distanceMatrix);
96  }
```

✓ **Description :**

Nous devons créer une fonction afin de libérer l'espace mémoire alloué à la matrice, ce qui nous permettra de nous assurer qu'il n'y a pas de fuites de

mémoire. Cela est essentiel pour maintenir l'efficacité du programme, en évitant de gaspiller des ressources système. En libérant correctement la mémoire après son utilisation, nous garantissons un fonctionnement optimisé du programme, permettant ainsi de maximiser ses performances et de prévenir toute surcharge inutile liée à une mauvaise gestion de la mémoire.

3.Fonction TSPUsingMatrix :

```c
#include "graph.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
void TSP(Graph* graph, int* path, int* fpath, long long* sum, long long* fsum, int flag, int n, int b, int a, int* sc) {
    if (flag == n) {
        long long currentSum = 0;

        for (int i = 1; i < n; i++) {
            int src = sc[path[i - 1]];
            int dest = sc[path[i]];
            int edgeCost = getEdgeCost(graph, src, dest);

            if (edgeCost == -1) return;
            currentSum += edgeCost;
        }
        int edgeCost = getEdgeCost(graph, sc[path[n - 1]], sc[path[0]]);
        if (edgeCost == -1) return;
        currentSum += edgeCost;
        if (currentSum < *fsum) {
            *fsum = currentSum;
            for (int i = 0; i < n; i++) {
                fpath[i] = sc[path[i]];
            }
        }
        return;
    }
    for (int i = a; i <= b; i++) {
        int temp = path[a];
        path[a] = path[i];
        path[i] = temp;

        TSP(graph, path, fpath, sum, fsum, flag + 1, n, b, a + 1, sc);

        temp = path[a];
        path[a] = path[i];
        path[i] = temp;
    }
}
int getEdgeCost(Graph* graph, int src, int dest) {
    for (int i = 0; i < graph->numEdges; i++) {
        if (graph->edges[i].src == src && graph->edges[i].dest == dest) {
            return graph->edges[i].weight;
        }
    }
    return -1;
}
void TSPUsingMatrix(int** distanceMatrix, int* path, int* fpath, long long* fsum, int flag, int n, int b, int a) {
```

```
48  void TSPUsingMatrix(int** distanceMatrix, int* path, int* fpath, long long* fsum, int flag, int n, int b, int a) {
49      if (flag == n) {
50          long long currentSum = 0;
51          for (int i = 1; i < n; i++) {
52              currentSum += distanceMatrix[path[i - 1]][path[i]];
53          }
54          currentSum += distanceMatrix[path[n - 1]][path[0]];
55          if (currentSum < *fsum) {
56              *fsum = currentSum;
57              for (int i = 0; i < n; i++) {
58                  fpath[i] = path[i] + 1;
59              }
60          }
61          return;
62      }
63      for (int i = a; i <= b; i++) {
64          int temp = path[a];
65          path[a] = path[i];
66          path[i] = temp;
67          TSPUsingMatrix(distanceMatrix, path, fpath, fsum, flag + 1, n, b, a + 1);
68          temp = path[a];
69          path[a] = path[i];
70          path[i] = temp;
71      }
72  }
```

✓ **Description :**

La création d'une nouvelle version du problème du voyageur de commerce (TSP) qui utilise la matrice générée à l'aide de l'algorithme de Dijkstra, laquelle stocke les poids (distances) entre les différentes villes, permet de réduire la complexité de la fonction. Grâce à cette matrice, les distances optimales entre les villes sont pré-calculées, ce qui évite de devoir recalculer ces valeurs à chaque étape du processus. Cela contribue à rendre l'algorithme plus rapide et plus efficace, en réduisant le temps de calcul global nécessaire pour résoudre le problème.

4.Updated main :

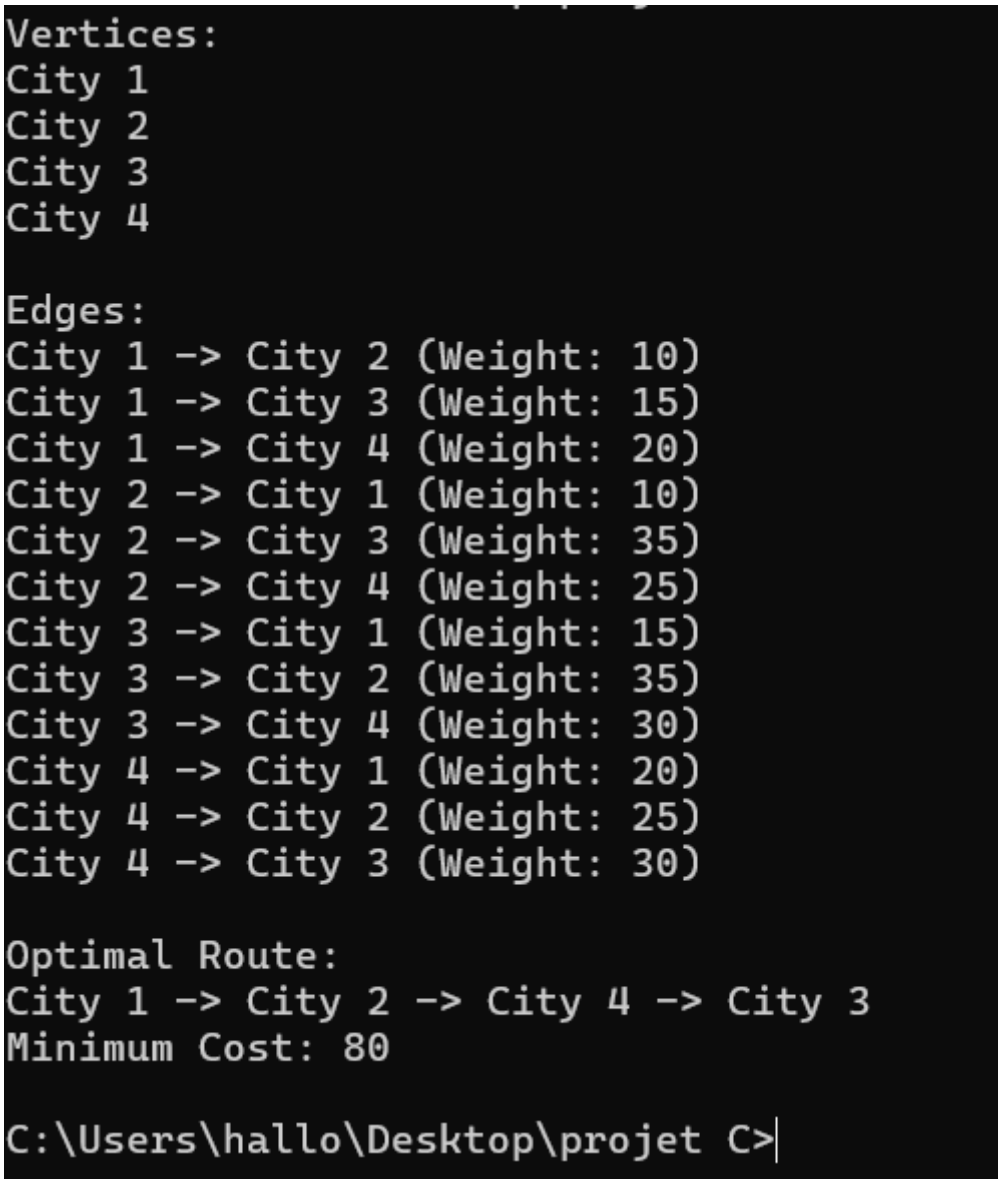
```c
#include "graph.h"
#include <limits.h>
#include <stdio.h>
#include<stdlib.h>

int main() {
    int n = 4;
    int numEdges = 12;
    Graph* graph = createGraph(n, numEdges);

    addEdge(graph, 1, 2, 10, 0);
    addEdge(graph, 1, 3, 15, 1);
    addEdge(graph, 1, 4, 20, 2);
    addEdge(graph, 2, 1, 10, 3);
    addEdge(graph, 2, 3, 35, 4);
    addEdge(graph, 2, 4, 25, 5);
    addEdge(graph, 3, 1, 15, 6);
    addEdge(graph, 3, 2, 35, 7);
    addEdge(graph, 3, 4, 30, 8);
    addEdge(graph, 4, 1, 20, 9);
    addEdge(graph, 4, 2, 25, 10);
    addEdge(graph, 4, 3, 30, 11);

    displayGraph(graph);

    int** distanceMatrix = createDistanceMatrix(graph);

    int path[n], fpath[n];
    long long fsum = LLONG_MAX;

    for (int i = 0; i < n; i++) {
        path[i] = i;
    }

    TSPUsingMatrix(distanceMatrix, path, fpath, &fsum, 0, n, n - 1, 0);

    printf("\nOptimal Route:\n");
    for (int i = 0; i < n; i++) {
        if(i!=n-1){
            printf("City %d -> ", fpath[i]);
        }else{printf("City %d", fpath[i]);
        }
    }
    printf("\n");
    printf("Minimum Cost: %lld\n", fsum);

    freeDistanceMatrix(distanceMatrix, n);
    freeGraph(graph);

    return 0;
}
```

✓ Description :

Nous avons dû mettre à jour la fonction principale afin d'inclure la nouvelle fonction et de traverser le chemin trouvé par l'algorithme du voyageur de commerce (TSP) pour afficher le chemin optimisé. Cela implique non seulement d'intégrer correctement la nouvelle fonction dans le flux principal du programme, mais aussi de suivre et de visualiser l'itinéraire optimal calculé par

le TSP. Cette mise à jour permet de présenter non seulement le coût minimal du voyage, mais aussi le trajet exact entre les différentes villes, garantissant ainsi que le programme fonctionne de manière complète et efficace.

5.Test de fonction :



✓ Description

Le code représente correctement le graphe, avec les villes et les connexions affichées avec leurs poids (distances).

L'itinéraire optimal trouvé par l'algorithme TSP est Ville 1 -> Ville 2 -> Ville 4 -> Ville 3 -> Ville 1, pour un coût total de 80.

Dans l'ensemble, le code démontre une implémentation réussie de l'algorithme TSP avec e Dijkstra algorithm pour ce graphe de villes.

Conclusion

Le code implémente avec succès l'algorithme TSP pour trouver l'itinéraire optimal. Cependant, il existe des défis liés à la complexité de l'algorithme.

✓ Problème de complexité

L'algorithme de force brute utilisé a une complexité en temps de O(n!), où n est le nombre de villes. Cela signifie que le temps d'exécution croît de manière exponentielle

avec le nombre de villes, rendant l'approche inapplicable pour des graphes de grande taille.

✓ Améliorations

Des améliorations futures pourraient inclure l'implémentation d'algorithmes TSP plus efficaces, tels que la méthode de Held-Karp ou des heuristiques, afin de réduire la complexité et permettre le traitement de graphes plus importants.

→ Malgré cette limitation, le code actuel reste une implémentation fonctionnelle pour des graphes de taille modérée. Des optimisations mineures, comme l'affichage des arêtes, pourraient aussi être envisagées.